Argonne
NATIONAL LABORATORY

Structured Hints:
Extracting and Abstracting Domain Expertise

Mathematics and Computer Science Division

Structured Hints:
Extracting and Abstracting Domain Expertise

by
M. Hereld,[1] R. Stevens,[1] T. Sterling,[2,3] and G.R. Gao[4]
[1]Mathematics and Computer Science Division, Argonne National Laboratory
[2]California Institute of Technology
[3]Louisiana State University
[4]University of Delaware

Contents

Structured Hints: Extracting and Abstracting Domain Expertise

Mark Hereld,[1] Rick Stevens,[1] Thomas Sterling,[23] and Guang R. Gao[4]

[1]Mathematics and Computer Science Division
Argonne National Laboratory
{hereld,stevens}@mcs.anl.gov

[2]Center for Advanced Computing Research
California Institute of Technology
tron@cacr.caltech.edu

[3]Department of Computer Science
Louisiana State University
tron@cct.lsu.edu

[4]Department of Electrical and Computer Engineering
University of Delaware
ggao@capsl.udel.edu

Abstract

We propose a new framework for providing information to help optimize domain-specific application codes. Its design addresses problems that derive from the widening gap between the domain problem statement by domain experts and the architectural details of new and future high-end computing systems. The design is particularly well suited to program execution models that incorporate dynamic adaptive methodologies for live tuning of program performance and resource utilization. This new framework, which we call "structured hints," couples a vocabulary of annotations to a suite of performance metrics. The immediate target is development of a process by which a domain expert describes characteristics of objects and methods in the application code that would not be readily apparent to the compiler; the domain expert provides further information about what quantities might provide the best indications of desirable effect; and the interactive preprocessor identifies potential opportunities for the domain expert to evaluate. Our development of these ideas is progressing in stages from case study, through manual implementation, to automatic or semi-automatic implementation. In this paper we discuss results from our case study, an examination of a large simulation of a neural network modeled after the neocortex.

1 Introduction

Current high-end computing (HEC) systems are extremely difficult to program. They lack system software and tools to support the high-level abstractions required for economical expression of advanced scientific and engineering simulations while producing high-performance code. As semiconductor technology has continued to evolve, both new opportunities and new challenges continue to emerge, demanding corresponding improvements to architecture. Most pronounced is the move to multicore components and the re-emergence of heterogeneous computing elements such as GPUs and ClearSpeed SIMD [2] attached processors. The IBM Cell architecture embodies both heterogeneous and multicore elements [5].

In an earlier paper [4] we described the *hierarchical threaded virtual machine* (HTVM) that defines a dynamic, multithreaded execution model, providing an architecture abstraction for HEC system software and tool development.

The complexity of these new architectures and the programming models that will support them has for some time been recognized as contributing to an expanding gap between domain science problem exposition and an effective response to the underlying architectural details of the system. Balancing the convenience and clarity of domain specific abstractions against the possible performance penalties induced by the relative inability of compilers to see through the abstraction is an aspect of the present challenge. An example from unstructured mesh-based computation has been described by White et al. [16].

Kennedy et al. [9] have developed methods for generating a component library version optimized for a target domain. Koes et al. [10] introduce an annotation-mediated interaction between the compiler and the application programmer to identify opportunities enabled when pointers are guaranteed to be independent. The programmer can supply `#PRAGMA INDEPENDENT` annotations to suggest optimization opportunities. Alternatively, the compiler can suggest candidates for the programmer to evaluate, including scores to indicate how much benefit might hinge on the

correctness of the assertion. A generalized approach to organizing potential dynamic optimizations has been added to the MIPSpro and gcc compilers [3]. In the highly constrained environment available to the embedded system, an interactive approach giving the programmer instantaneous performance information and detailed control in real time of optimization strategies has proved effective [11]. One hiding place for opportunities for performance improvements is behind the facade of domain-specific abstractions. A method for exposing these potential opportunities to the compiler so that it can apply its suite of optimizations has been reported by Quinlan [13].

A growing number of research projects are targeting development of large-scale simulations of biological neural networks [8, 1, 12, 6]. Goals and approaches vary widely, but the underlying need for efficient execution of complex code is shared.

In this paper we discuss a method to capture and convey domain expert knowledge to the compiler where it can be exploited to generate optimized codes. The captured knowledge also informs a runtime performance monitoring mechanism to support the continuous compilation component of the HTVM. We discuss the idea, our case study of a simulator of large neural networks, and opportunities and future work.

2 Approach

The gap is widening between expressions of domain-specific computations and expressions tailored to efficient execution on a given system architecture. Domain experts are and will continue to be challenged to write high-performance codes. The approach described in this section is to build an interactive preprocessor that captures information from the domain expert to be used in optimizing code fragments, determining runtime monitoring priorities, and driving an adaptive compiler – all components of the HTVM system architecture.

2.1 Domain-Specific Knowledge

It is important to develop a sense of what the domain expert can know about the application's behavior without getting into architecture and system-dependent expertise. This is a difficult task because application development for high-performance computing has always required expertise at both ends of this spectrum – a requirement that is fast becoming impossible to reasonably require. In fact, languages typically expose a great deal of superfluous flexibility that the compiler then often has to reclaim in the name of optimization. Loop transformations are one example.

An informal summary of our thinking on what the domain expert can reasonably be expected to understand in-

cludes the intent of the simulation, aspects of the expected behavior of the simulation and its components, and properties of the high-level program describing the simulation. These high-level observations can be expanded into more-immediate questions that the domain expert might be able to answer. Are the data objects fixed or likely to be destroyed, moved, or extended? Can the internal components of a data object be rearranged by the compiler without "breaking the program"? Are there typical limits to the length of dynamically managed lists and queues? Will this value change rapidly or slowly for the most part? These questions are the sort that the domain expert can often readily answer.

On the other hand, the compiler often cannot easily answer such questions using static analyses. Moreover, the number of quantities to track often make unguided dynamic techniques intractable.

One possible solution is a paradigm where structs laid out in C by the domain expert are interpreted by the compiler as suggestions only. The information extracted from the domain expert is used to suggest to the compiler and runtime system means for dynamically optimizing the performance of the application. They are hints, not demands. Without reference to the underlying hardware architecture or HTVM software architecture, the hints must address issues of data locality, monitoring priorities, data access patterns, and computation patterns. These will be mapped directly to specific actions, weighting schemes, and optimization strategies in the HTVM system software. To capture, express, and implement this idea, we are exploring the following component mechanisms:

- Source code <u>annotations</u> provided by the domain expert (pragma statements will typically suffice) will provide hints to the compiler.

- Where one of the usual <u>metrics</u> (e.g., performance, memory use) might be undesirable as the primary indicator, the domain expert can provide alternatives to associate with a given dynamic optimization.

- These annotations will also be informed by the <u>interactive dialog</u> established between the domain expert and the preprocessor.

- Where possible, the compiler (or other parts of the system) can tentatively identify its own <u>candidates for ratification</u> by the domain expert (as in Koes et al. [10]).

- The structured hints accumulated in this way will be available for <u>reuse</u> within the domain and exploitation by translation to other domains.

The resulting expertly culled guide to optimization, the structured hints, includes data structures, dependencies, weights, and rules. In addition to focusing the compiler's attempts at optimization, these structured hints will be an

integrated part of our Structured Hints Database, providing the runtime system with an informed and tailored set of options with which to make its choices.

Each hint can be targeted at some part of the execution model: the adaptive compiler, runtime system, or monitoring system. For example, informed choices about which pieces of the code to instrument, and how, will become part of the metric suite used by the adaptive compiling and runtime system to adjust resource allocation and compilation strategy during execution. As another example, the domain expert can identify critical parameters to be adjusted by the compiler for its adaptive optimizations, thereby narrowing the parameter space to be searched.

2.2 Interactive Preprocessor

In this section we outline the proposed process for interactively collecting hints from the domain expert. We envision the interaction between the domain expert and the preprocessor as an iterative conversation managed by the preprocessing engine.

Structured hints will be represented in a knowledgebase populated by the results of the interactions between the application programming, domain expert, and compilation tools. It will be used by various stages of the code translation process, the HTVM system, and the runtime monitoring system.

The adaptive compile and runtime system of the HTVM will require feedback derived from the execution and resource allocation monitoring. The hints will drive both static and dynamic optimizations of the program execution. They will provide the system with guidance on degrees of freedom most likely to affect performance, likely bottlenecks in the code, unpredictable aspects of data locality, and computational work patterns to steer monitoring.

Figure 1 illustrates the iterative process that defines the interaction between domain expert and preprocessor. The Domain Expert is guided by the Positor to provide expert understanding of the properties of the domain-specific objects (structures and methods) that allow the Annotator to generate a new version of the source code. The Evaluator analyzes each iteration and may suggest additional opportunities to the Domain Expert by producing a newly annotated source version that the Positor uses to guide the Domain Expert. A final version of the source is then passed to the Translator for conversion into HTVM targeted code.

The first round of interaction with the Positor is driven by a template – initially generic, but colored over time by knowledge in the Structured Hints Database – of directed questions about each instance of a data structure and each recognized access pattern. The savvy Domain Expert may prune this process by providing annotations in the initial version of the source.
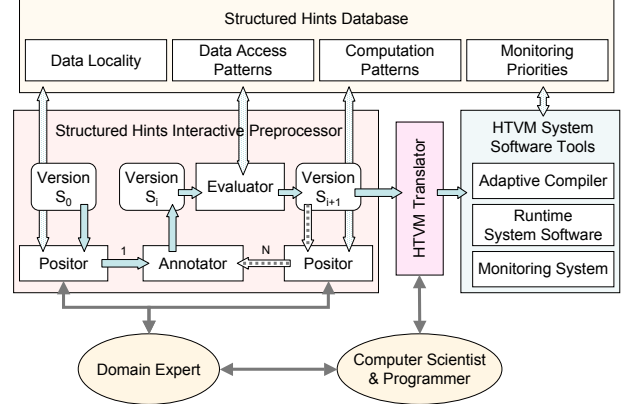
Figure 1. Structured hints workflow integrated into the HTVM architecture.

3 Parallel Simulation of Neural Networks

For the current study we focus on one application code, pNeocortex. This section briefly describes the basic properties of its data model and algorithm.

The pNeocortex simulation [7, 15] uses biologically based compartmental neuron models connected probabilistically to form a dense network. Each neuron might connect to thousands of others according to the thirty probabilistic rules that constitute the wiring diagram. The model includes four basic neuron types. The neuron types and wiring rules derive from the literature describing the neurophysiology of the neocortex.

The compartmental model is a style of describing neurons that enables the domain scientist to build up a detailed neuron from a modest set of component types. Basic compartments are linked together in branching chains to form dendrites. This basic compartment object encapsulates the state of the Hodgkin-Huxley model segment and is acted on by methods that initialize these values, prepare them for each simulation step, facilitate propagation of signal data, and evolve the state forward in time. Other compartment objects are specialized to represent ion channels and spiking elements.

The style of model building represented by this approach has many advantages for the domain expert. It is modularized in familiar units, amenable to an economical hierarchical description of a complex system, flexible, and consequently powerful.

The model can be thought of as lists of compartments and of connections. The pseudocode for pNeocortex (Figure 2) shows that all types are acted on as a group by the INIT() method. Subsequently, members of each type –

```
while ( TIME < TOTAL_SIM_TIME )
do {
  foreach Object in MODEL
    { Self.INIT(); }
  Synchronize;
  ExchangeData;

  foreach SpikingObject in MODEL {
    if (potential >= threshhold)
      foreach SynapticConnection in Self
        { Target.RegisterEvent(); }
  }
  foreach ChannelObject in MODEL {
    foreach ContactPotential in Self
      { Self.Adjust(V); }
    Self.Integrate();
  }
  foreach CompartmentObject in MODEL {
    foreach ContactPotential in Self
      { Self.Adjust(Vm, Rm); }
    Self.Integrate();
  }
  Synchronize;
  step TIME;
}
```

Figure 2. Pseudocode for pNeocortex.

SpikingObject, ChannelObject, and the generic CompartmentObject – are acted on in turn with various methods specific to their role in the simulation. Within each of these bulk operations there is no constraint on execution order. On the other hand, the ordering of foreach loops expresses the scheduling of the simulation phases that must be preserved.

These tasks and the data objects that they act on are distributed by assigning groups of neurons to each processor. Interprocess communication consists almost entirely of propagating spike events.

4 Case Study and Results

In our study of this code and the current state of large-scale neural network simulation codes, we have tried to find opportunities for an expert in the science domain (with experience in modeling and simulation) to make observations about the typical evolution of the signals, the intent of the model abstractions, and the constraints on the ordering of evaluations.

The pNeocortex simulation presents an interesting example. It has a challenging communication pattern. It might be partitioned at many granularities – compartment, neuron, or tissue slice – with different implications for performance. Some aspects of such simulations are well studied, offering

opportunities for improving performance against which our methods can be tested. Moreover, new and unproven opportunities arise from new architectural and program execution models.

As implemented, pNeocortex makes extensive use of linked lists to build both the neurons and the synaptic connections. This idiom maps well to the incremental and irregular process needed to construct the model but carries with it severe penalties at execution time. In many cases, access through pointers hampers the compiler's ability to determine dependencies. Several observations can be made by the domain expert that could lead to profitable source code translations. These and their possible implications for optimization opportunities are as follows:

- **Static Linked Lists** Some of these object lists are never reordered once the model has been built and initialized. *The compiler could reinterpret the model creation semantics with the domain expert's asserted guarantee that the final list is never reordered, refactored, truncated, or augmented.*

- **Opaque Structures** For some object types, data access is strictly regulated by an opaque interface – improving the chances that automatic source translation is tractable. *Internal data could be reorganized in any way without changing the execution results. In fact, the entire list of structs could be refactored into a struct of lists if the compiler saw an opportunity for better performance or better memory use.*

- **Short Queues** Although variable in length, the event queue maintained by each ChannelObject (which is scanned during the ChannelObject.Integrate() action) is typically short. Although the domain expert cannot guarantee a maximum, he could say that the length is virtually always less than 3, for example. *The compiler could choose to implement the queue in a way that leverages this fact.*

- **Predictable Behavior** The voltage potential in any compartment of the system is likely to be characterized over much of the simulation as mostly slowly varying, infrequently punctuated by spikes or bursts of spikes. *This observation could enable the compiler and runtime system to employ any of several adaptive techniques to lower the computational burden of forward integration.*

- **Duplicated Object Data** The memory required by the neurons is dwarfed by that required to represent the interconnections. *There may be advantages to maintaining and evolving duplicate copies of neuronal state data across many nodes in order to lessen the impact of communication.*

- **Low Event Rate** Neurons are often quiet, posting spike events infrequently. *Depending on how many*

4

neurons are in a thread and the cost of communicating spike events to other threads, predictive methods might be employed to reduce the overhead in checking for events.

As a specific example of what might be done in the case of static linked lists, consider the basic semantics of building a large list of objects. If the list was implemented as a linked list, the compiler could substitute equivalent code to implement a linked list of subarrays with the possible advantages of reduced fragmentation, better memory hierarchy performance, and new opportunities for vectorization. Shape analysis [14] might be used by the compiler to identify the opportunity or as part of a verification step. The domain expert should be enabled by the tools to express the concepts in a natural way, without unnecessary stylistic obligations and burdensome syntactic requirements.

```
//        simple data structure
struct linkedType {
  linkedType* next;
  double value;
}
//        object creation
myPtr = (linkedType *)
        malloc( sizeof(linkedType) );
//          adding the object to the list
myPtr->next = list->next;
list = myPtr;
//        traverse the list
myPtr =list;
while (myPtr != NULL) {
  // do stuff
  myPtr = myPtr->next;
}
```

(a) simple linked list

```
//        chunky data structure
struct chunkyType {
  chunkyType* next;
  int objectcount;
  double value[N];
}
```

(b) linked list chunks

Figure 3. Comparing the semantics of model building implementations.

Figure 3 shows an example of the kinds of references to a singly linked list that would typify a simple implementation used repeatedly in pNeocortex. A linked list of N-element sublists was manually transformed to one of the linked lists created during model construction. N was chosen to be 100

for this test. The new data type is similar to the ArrayList in Java.

As a result of this transformation on a structure used in the setup phase of the simulation, modest gains in memory allocation efficiency, resulting in a 10% reduction in the footprint of the application, enabled slightly larger problems to be run in these tests. More significant, a performance speedup was realized during this phase – between 27% and 55%.

5   Conclusions

In this paper we have discussed an approach to bridging the gap between the domain scientist expertise and the details needed to refine a simulation code for optimized performance on HEC systems. The approach is designed with the adaptive HTVM execution model in mind. The approach exploits opportunities for improving performance made visible to translation engines by hints supplied by the domain expert.

Much work is needed to move this idea forward. We plan to pursue the following steps:

- Carry out additional manual translation experiments, drawing on the opportunities described in the previous section.

- Build and test scripts to automate the translation process and to expose vulnerabilities in the translation, leveraging existing source transformation tools where practical.

- Derive elements of the sufficiently robust hinting vocabulary from the experiences gained in the previous steps.

- Design the structured hints database.

- Develop linkages between HTVM monitoring interface and the structured hints database.

- Characterize application performance in the adaptive runtime environment of HTVM.

Our ultimate goal is a flexible and extensible bridge between the domain expertise and the HTVM execution model. This bridge will be cast in the form of a methodology for creating annotations, scripted support mechanisms, and a knowledge base that captures the domain expert's advice in a way that can be used by the adaptive compiler and runtime to monitor and optimize performance.

References

[1] R. Ananthanarayanan and D. S. Modha. Anatomy of a cortical simulator. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, Reno, Nov. 2007.

[2] ClearSpeed. ClearSpeed CSX600. `http://www.clearspeed.com/`.

[3] G. Fursin and A. Cohen. Building a practical iterative interactive compiler. In *1st Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'07), colocated with HiPEAC 2007 conference*, January 2007.

[4] G. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. Hierarchical multithreading: programming model and system software. In *20th International Conference on Parallel and Distributed Processing Symposium*, 2006.

[5] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[6] M. Hereld, R. L. Stevens, J. Teller, and W. van Drongelen. Large neural simulations on large parallel computers. *Int. J. Bioelectromagnetism*, 7:44–46, 2005.

[7] M. Hereld, R. L. Stevens, W. van Drongelen, and H. C. Lee. Developing a petascale neural simulation. *Proceedings of the 26th Annual International Conference of the Engineering in Medicine and Biology Society*, 6:3999–4002, 2004.

[8] C. Johansson and A. Lansner. Towards cortex sized artificial neural systems. *Neural Networks*, 20:48–61, 2007.

[9] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, and L. Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803–1826, 2001.

[10] D. Koes, M. Budiu, and G. Venkataramani. Programmer specified pointer independence. In *Proc. Workshop on Memory System Performance*, pages 51–59, Washington, D.C, 2004.

[11] P. Kulkarni, W. Zhao, S. Hines, D. Whalley, X. Yuan, R. van Engelen, K. Gallivan, J. Hiser, J. Davidson, B. Cai, M. Bailey, H. Moon, K. Cho, and Y. Paek. Vista: Vpo interactive system for tuning applications. *Trans. on Embedded Computing Sys*, 5:819–863, 2006.

[12] H. Markram. The blue brain project. *Nat. Rev. Neurosci*, 7:153–60, 2006.

[13] D. Quinlan, M. Schordan, R. Vuduc, and Q. Yi. Annotating user-defined abstractions for optimization. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8 pp., 2006.

[14] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[15] W. van Drongelen, H. C. Lee, M. Hereld, D. Jones, M. Cohoon, F. Elsen, M. E. Papka, and R. L. Stevens. Simulation of neocortical epileptiform activity using parallel computing. *Neurocomputing*, 58-60:1203–1209, June 2004.

[16] B. S. White, S. A. McKee, B. R. de Supinski, B. Miller, D. Quinlan, and M. Schulz. Improving the computational intensity of unstructured mesh applications. In *Proceedings of the 19th Annual International Conference on Supercomputing*, Cambridge, Mass., pages 341-350, 2005.